A Cross-Virtual Machine Network Channel Attack via Mirroring and TAP Impersonation

Atif Saeed, Peter Garraghan
School of Computing and Communications
Lancaster University
{a.saeed2, p.garraghan}@lancaster.ac.uk

Barnaby Craggs, Dirk van der Linden, Awais Rashid
Department of Computer Science
University of Bristol
{barney.craggs, dirk.vanderlinden, awais.rashid}@bristol.ac.uk

Syed Asad Hussain
Department of Computer Science,
COMSATS Inst. of Information Tech.
asadhussain@ciitlahore.edu.pk

***Abstract*—Data privacy and security is a leading concern for providers and customers of cloud computing, where Virtual Machines (VMs) can co-reside within the same underlying physical machine. Side channel attacks within multi-tenant virtualized cloud environments are an established problem, where attackers are able to monitor and exfiltrate data from co-resident VMs. Virtualization services have attempted to mitigate such attacks by preventing VM-to-VM interference on shared hardware by providing logical resource isolation between co-located VMs via an internal virtual network. However, such approaches are also insecure, with attackers capable of performing network channel attacks which bypass mitigation strategies using vectors such as ARP Spoofing, TCP/IP steganography, and DNS poisoning.**

In this paper we identify a new vulnerability within the internal cloud virtual network, showing that through a combination of TAP impersonation and mirroring, a malicious VM can successfully redirect and monitor network traffic of VMs co-located within the same physical machine. We demonstrate the feasibility of this attack in a prominent cloud platform – OpenStack – under various security requirements and system conditions, and propose countermeasures for mitigation.

I. Introduction

Virtualization technologies such as Xen, KVM, and VMWare are fundamental components in cloud computing, and are used to run multiple Virtual Machines (VMs) on a single physical machine. Virtualization provides strong *logical isolation* of resources for VMs located on the same physical machine ensuring they cannot interfere with the operation of other co-resident VMs [1]. Implemented by the Virtual Machine Monitor (VMM), or hypervisor, logical isolation creates an internal virtual network to separate, or isolate, logical networks within a shared physical network. Logical isolation is an important security feature of cloud computing to prevent exploitation of co-resident VMs by a malicious VM, by providing protection against poorly designed or ineffective access-control policies [2]. This also prevents co-residing VMs from exfiltrating data and interfering with each other's execution.

However, as co-residing VMs share their underlying VMM, virtual network and hardware, they are susceptible to cross-VM attacks. Studies have demonstrated co-resident vulnerabilities such as shared file systems [3], cache side-channels [4], [5], and compromised VMM via rootkits [6]. Moreover, a malicious VM might potentially access other VMs through network connections, shared memory, and other shared resources [7]. For example, a bridge configuration can be exploited to create a virtual hub, used by all VMs, to communicate over the network. This allows a malicious VM to *sniff* virtual networks using tools such as Wireshark [8].

Despite the potential threat of cross-VM attacks through exploitation of shared memory and disk space, there has been no demonstration of a fine-grained cross-VM attack using the network channel. Current network-based attacks involve exploiting existing vulnerabilities in networking technologies (e.g., ARP spoofing, DNS poisoning), but are difficult or impossible to utilize for cross-VM targeted attacks due to additional layers of isolation between co-residing VM architectures designed to mitigate such attacks (e.g., Neutron [9]).

In this paper we present a new cross-VM network channel attack that creates data leakage to passively monitor the network traffic of target VMs by exploiting the virtual internal networking of open virtual switch. A malicious VM redirects the network traffic of target VMs to a specific destination by impersonating the Virtual Network Interface Controller (VNIC), and enables a mirror within the bridge of the network controller where the target VM is currently sending network traffic. The demonstrated attack can extract decrypted information from target VMs by using open source decryption tools such as aircrack [10].

We demonstrate the feasibility of our attack within OpenStack using a laboratory testbed environment, measuring respective network traffic. Our findings indicate that the attack successfully exposes noticeable data leakage vulnerabilities. Moreover, by exploiting standard assumptions of provisioning cloud services, it is possible to hide the malicious activity of attacking VMs as seemingly normal activity—both jeopardizing the privacy, as well as predicting target VM activity. For responsible disclosure, all vulnerabilities found in our research have been reported to the OpenStack and Ravello security teams, where we have also provided solutions for fixing identified issues.

The rest of this paper is organized as follows: Section II describes the background of cross-VM attacks. The attack scenario and stages are detailed in Section III. Section IV presents the attack scenario experiment scenario, followed by the evaluation in Section V. Section VI proposes attack countermeasures and Section VII provides concluding remarks.

II. Background

Cross-VM attacks and their ability to exploit shared resources to extract or leak the sensitive information from a target VM have been investigated in numerous ways. An established assumption was that co-located VMs sharing network interfaces are able to trust each other [11]. However this was shown to be a weak assumption when considering the sharing of physical hardware in public clouds and possibility of co-residency attacks [12].

Zhang et al. [13] categorize co-residency attacks into three distinct side channel classes: (1) Access-driven side channel that exploits shared micro-architectural modules like caches, (2) time-driven side channel, only possible when the total execution time of cryptographic operations with a fixed key are influenced by the key value, and (3) trace-driven to capture a profile of cache activity.

A. Related Work

Researchers have identified a number of cross-VM attacks as shown in Table I. Within cross-VM Address Resolution Protocol (*ARP*) attacks [11], the attacking VM launches an *ARP* spoofing attack by forging an identical IP address within the target VM, and sends an *ARP* to the virtual router. The virtual router updates the routing table when the spoofed *ARP* is received. As a result, any traffic directed to a target VM is sent instead to the attacking VM, which can then decide to either perform sniffing or modification.

In bridge network configuration mode [11], the bridge acts as a virtual hub. All VMs share the virtual hub to communicate with the network. An attacking VM is able to sniff the virtual network by using a sniffing tool, such as Wireshark [8]. In the router network mode [11], a router plays a role of a virtual switch using a dedicated virtual interface to connect to each VM. Here, a malicious VM can undertake ARP poisoning [14], redirecting packets towards themself, and then sniffing packets going to and coming from other VMs.

Works such as [15], [16] demonstrate methods to leak sensitive data through TCP/IP. Ranjith *et al.* [17] provides a method for using timing channel for data leakage.

TABLE I
Comparison of Related Work

Attack	*Description*	*Ref.*
Side Channel	Time-driven	[13]
	Access-driven	[13]
	Trace-driven	[13]
Covert Channel	TCP/IP Steganography	[15]
	TCP/IP header Steganography	[16]
	Timing Channel	[17]
DoS	Illegal use of resources	[11], [18], [19]
Network Channel	ARP Poisoning	[11], [14]
	Sniffing	[11]
	Spoofing	[11]

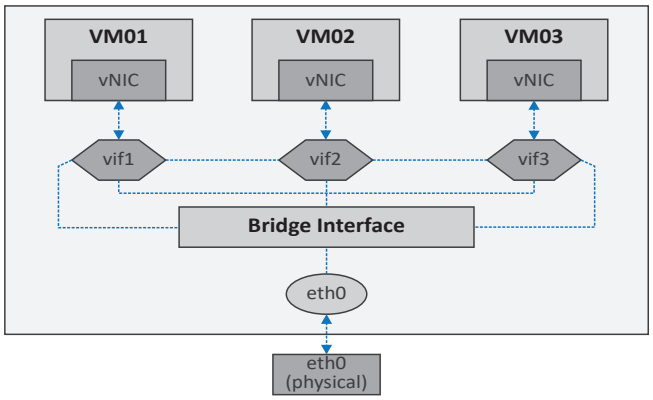


Fig. 1. Virtual bridge general architecture

B. Proposed Attack Vector

To the best of our knowledge, no previous work has achieved redirection of a co-residing target VM's network traffic, by exploiting the network channel through a combination of impersonation and mirroring. Impersonation is the first step in the attack vector that makes it possible for the attacker to penetrate the system, and mirroring redirects a copy of the target VM's network traffic to a selected destination point. We believe that a combination of these approaches makes it possible to redirect target VM network traffic at run-time. Even encrypted network traffic, if successfully intercepted, may be readily decrypted [20] using third-party tools such as aircrack [10]. The decrypted traffic of the target VM can be used for further analysis to predict its activity, and compromise privacy.

C. System Model

This section describes the core components of the cloud computing network architecture and implementation in OpenStack.

1) General Architecture

Within the cloud computing network architecture, VMs are connected via a virtual bridge residing in the VMM of each physical machine (see Figure 1). The virtual bridge has a virtual interface (*vif*) connecting the Virtual Network Interface Card (VNIC) of each VM, which is sent to the physical machine's Ethernet device connected to an external network such as the internet. The virtual bridge also supports the internal communication between VMs, allowing the cloud administrator to assign and manage VM network resource usage and traffic in order to adhere to specified Quality of Service (QoS). To construct a new virtual network, a new virtual bridge has to be instantiated within the physical server (such as *bridge0*), and a unique IP address aassigned to each *vif* which connects to the VM's virtual Ethernet card.

2) OpenStack Components

OpenStack is used to create virtual machines and resources through an Infrastructure-as-a-Service (IaaS) model. At a high level of abstraction, it consists of three main components:

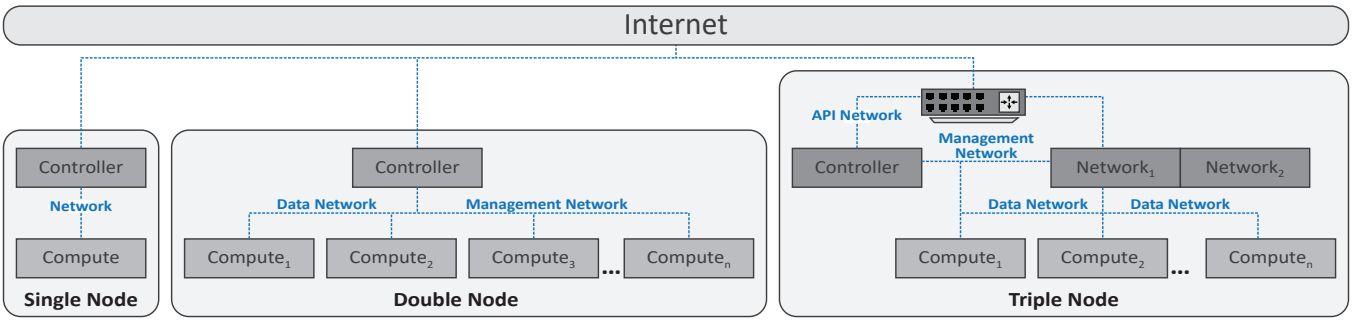


Fig. 2. Single, double, and triple node setups.

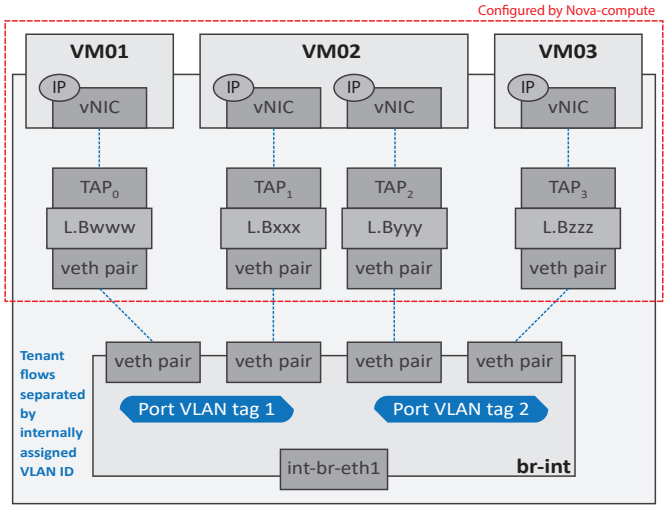
1) The **Controller** component is responsible for executing the management software's services, needed for the OpenStack platform to run. It runs, among others, the identity service, image service, compute and networking management, networking agents, network scheduler, and message queue.
2) The **Compute** component is responsible for executing VMs' instances within the system. Additionally, it is responsible for provisioning firewall services for each physical machine, and facilitates networking service agents connecting VMs to the virtual network. The system may include multiple compute components.
3) The **Network** component is responsible for running networking services such as Neutron, L3, metadata, DHCP, and Open vSwitch. It manages all communication between other OpenStack components, VM networking, and routing. Networking services such as DHCP and floating IPs allow instances to connect to public networks.

These components can be deployed in a single machine or separate nodes (i.e., physical machines) to create single-node and multi-node setups (see Figure 2). In a single node setup, all components reside within the same physical machine for provisioning all VM and networking capability. In a multi-node setup, a single controller (i.e., a physical machine hosting the controller component) is responsible for system management of all compute nodes executing VMs.

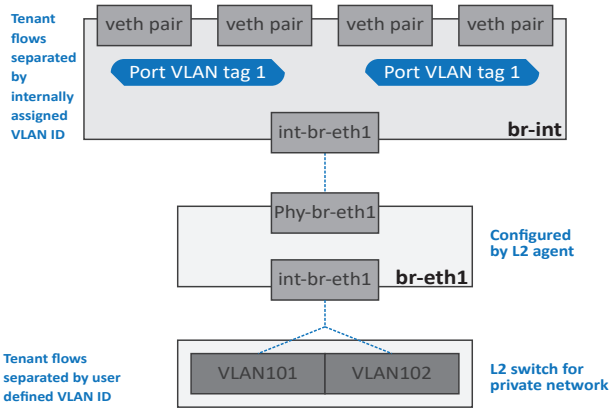
3) OpenStack Networking

OpenStack provides security groups and policies that can be applied to parameters on incoming and outgoing user traffic, VMs, and containers. Security groups, such as firewall rules, SSH control, and port bounding can be defined and applied dynamically to improve protection. The flexibility of security groups allows enterprises to quickly respond to threats. The network architecture of OpenStack (shown in Figure 3(a)) is implemented through four virtual networking devices:

- The **Test Access Point (TAP)** is an external monitoring device between the physical Ethernet card and each VM within the physical machine.
- The **Veth Pair** is a virtual network cable connecting the linux bridge to a virtual bridge residing in a physical machine.
- The **Open Virtual Switch (OVS)** is the virtual bridge responsible for managing incoming and outgoing traffic. It functions as a Layer-2 network switch providing different features of Access Control List and Virtual LAN (VLAN), as well as providing subnet or private network functionality to VMs. Each VM has an IP address visible to the virtual switch and is connected to a OVS bridge interface. The



(a) Networking devices



(b) Function of br-int and br-eth

Fig. 3. OpenStack networking interfaces

OVS contains two sub-components: (1) *br-int*, responsible for VLAN tagging (assigning the ID network traffic of VMs) and used by cloud providers to implement isolation between VMs, and (2) *br-ext*, used to bridge the virtual bridge to the physical network device.

- The **Linux bridge (L.B)** is responsible for communication between the *br-int* and each VM's *TAP*, recording it through a MAC caching table that saves the address and port number of packets between the VM and the Ethernet card. This is used to prevent packets of unknown IPs flooding to all VMs.

A VM creates and stores data on the associated VNIC, such as *eth0*. The data is then transmitted to the *TAP* on the compute host. Normally, a *TAP* offers an access path to data passing through a network. The TAPs are further linked to the Linux bridges that pass the data to *Veth Pair* which acts as one side of the cable. Data sent to one side of *Veth Pair* can be received at the other end. The other end of the pair is on the integration bridge: *br-int*. This bridge is responsible for the attachment of all the VM's *TAP*s and any other bridge on the system. The integration bridge further connects with *br-eth* as shown in Figure 3(b).

III. Attack Stages

This section describes assumptions and stages required to perform a cross-VM network channel attack, focusing on the OpenStack cloud platform [21] as a case study.

The feasibility of our proposed attack assumes that an attacker is capable of achieving control of a VM residing on same physical machine as the target VM [4]. Achieving this control has been demonstrated within prior research by using a network-based method to launch a co-location attack within a public cloud such as Amazon EC2 [4]. An attacker is capable of launching many VM instances within the same geographical region as the target VM, and use different methods to determine whether a VM is co-located successfully. This can include methods such as:

1) Execute a TCP SYN traceroute to detect the first hop of network traffic (e.g., the Dom0 in the host Xen server) between attacker and target VM. This results in an identical Dom0 IP address indicating successful co-location.
2) Calculate network packet round-trip time between attacker and target VM. A smaller value indicates that the two VMs share the same machine.
3) Check internal IP addresses of attacker and target VM. Numerically close internal IP addresses indicate the two VMs are more likely to be on the same server.

Our attack setting is based on virtual network solutions within public clouds which co-locate VMs in the physical machines using the hypervisor. Another use case separates OSs into multiple components with distinct privilege levels that are isolated by virtualization [22], [23], such as Qubes [24], an open source operating system running as multiple virtual machines on the hypervisor.

We target systems with modern multicore processors used in public clouds. We assume that the attacker and target are on separate network domains, each are assigned a number of disjoint resources such as virtual CPUs (VCPUs), VLANs, and virtual storage. All VMs are assigned the same privilege levels [25]. We further assume that OpenStack ensures logical isolation between mutually untrusted co-resident VMs, and that the attacker is unable to exploit software vulnerabilities permitting them to take control of the entire physical machine. The attack we propose, therefore, uses the cross-VM network channel to redirect the network traffic of target VMs. Constructing such a network channel requires multiple stages within a cross-VM setting, as depicted in Figure 4.

***Stage 1**: Observe Current Network Architecture*

OpenStack's default security settings prevent spying on target VM network traffic, particularly when attacker and target VMs reside on separate network domains.

The ability to spy on target network traffic by exploiting the virtual network has been successfully performed within non-virtualized settings. However, we do not utilize this exploit because OpenStack's default security perimeter (Firewall as a Service (FWaaS)) continuously monitors unusual activity, and prevents attachment of any external devices by blocking direct access to the system. The main objective is therefore to attempt to directly access (or become part of) the current system.

The first step is to identify and exploit a point of entry into the system to launch the attack. This can be performed by creating a dummy network interface for devices which do not have an active NIC adapter, and are not part of the running virtual network.

A dummy interface is typically used to avoid network communication from down state, creating a virtual 'stub' in the 'UP' state where IP addresses can be assigned not bound to a physical interface. This device serves as an alternative to the loopback interface in a system. If the network is disconnected, the loopback interface is the only way to communicate, in which case the physical machine IP address should be added as an entry in the host routing, and assigned to the dummy interface that is delivered locally.

The dummy network interface needs to be connected to other devices to exploit OpenStack's security perimeter and become part of the current OpenStack system. This is implemented within OpenStack by executing the following command:

```
ip link add dummy0 type dummy
```

***Stage 2**: Network Interface Impersonation*

In the event of OpenStack detecting that the real network card is activated, the next challenge is to mislead OpenStack's security perimeter by converting the identity of the regular interface into a TAP, which acts as a normal, valid device in network system.

Impersonating the regular interface to TAP is necessary to ensure we can connect our device to the network. The TAP acts as an inactive device as it does not interfere in any running network settings, and is compatible with the running network configuration. The attacker VM now has two interfaces: (1) a normal Ethernet card, impersonating a TAP with no valid

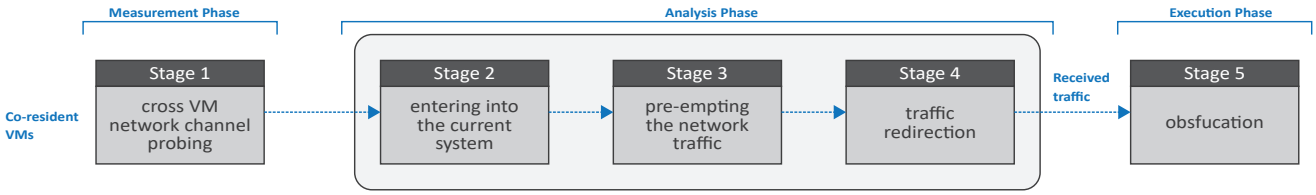


Fig. 4. Main steps in network channel attack

identity, and (2) the dummy interface. Next, a connectivity request has to be sent to the Linux bridge, which presumes it to be a valid TAP, and adds it using a method similar to the adding TAPs conventionally. After successfully completing this phase, the attacker can now penetrate into the network. This impersonation can be performed within OpenStack by:

Manually removing all types of interface identity in the network configuration file at

```
/etc/network/interfaces
```

and then restarting networking services by executing

```
/etc/init.d/networking restart
```

that makes this change persistent within the system file. This results in a network device without a valid identity functioning like a TAP. The difference between TAP and regular network device can be seen in Figure 5. *enp0s3* is the regular network device and *test0* is the tap device which has no valid identity.

***Stage 3**: Network Traffic Observation*

To spy on the network traffic of a target VM unobtrusively, we can direct the network traffic to a specified destination port. This entails determining who is responsible for redirecting the real time network traffic, and placement so as to hide from others. This is achieved by creating a mirror - a Linux tool with the ability to redirect traffic from one port to another - and placing the mirror at the internal interface of the network bridge that all other VMs communicate through.

Existing schemes [11] are unable to provide assistance in redirecting network traffic by using a combination of mirroring and impersonation. This stage can be performed in OpenStack by executing the following steps at the bridge, setting the dummy interface as a destination port:

```
test0     Link encap:Ethernet  HWaddr ca:50:2c:ba:4f:58
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

[redacted]-VirtualBox:~$ ifconfig enp0s3
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:ce:8e:88
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fece:8e88/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:344498 errors:0 dropped:0 overruns:0 frame:0
          TX packets:160039 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:288569674 (288.5 MB)  TX bytes:10398218 (10.3 MB)
```

Fig. 5. Difference Between TAP and eth0

```
create Mirror name=<mirror_name>
select-src-port=@br-int && set mirror @ br-int
```

***Stage 4**: Traffic Redirection at Destination Point*

If we observe the network traffic of a target VM at an open network location, it is likely that an attacker's activity will be monitored by the security perimeter of OpenStack cloud, which would subsequently block the attacking VM. It is possible to overcome this challenge by redirecting the target VM network traffic at a set hidden destination point. When it passes through the network bridge which contains a mirror discussed in Stage 3, the network traffic will be redirected from the internal bridge port towards our set destination port. This will result in not only target VMs but also the security perimeter being unaware of network traffic redirection. This redirection is performed within OpenStack by the following commands:

```
select-src-port=@br-int  &&  select-dst-port=@dummy0
```

***Stage 5**: Obfuscation*

The last stage entails eliminating any footprints of the attack by hiding all used devices and routes from the Linux tool that shows the static route and interfaces. This can be performed using network monitoring tools such as route n. After having attached the impersonated interface to *br-int*, a network log cleaner can be used, which will remove its identity and remain invisible in the route tool.

Combining all these stages together allows us to launch a cross-VM network attack as depicted in Fig. 6. The vulnerability that we exploit for this attack is Cloud provider's permission to bridge a TAP interface with no private Ethernet at the back end which is used to access the Internet.

IV. Experiment Setup

Attack scenario: The scenario we use to demonstrate our attack is for an attacker VM (VM1) to intercept and spy on communication between two communicating target VMs (VM2 and VM3). The placement of the mirror can then be used to perform redirection of target traffic.

We performed our attack within OpenStack using multiple node configurations encompassing single node, double node, and triple node. We deployed three guest VMs within a physical machine (Intel Core Z Q9650 @ 3.0Ghz) using the KVM hypervisor. VM1 was configured to be the attacker VM, while VM2 and VM3 are targets. Each VM is configured with two vCPUs different OS including Ubuntu 15.10(VM1), cirros (VM2), Windows 10 (VM3), and configured with both floating

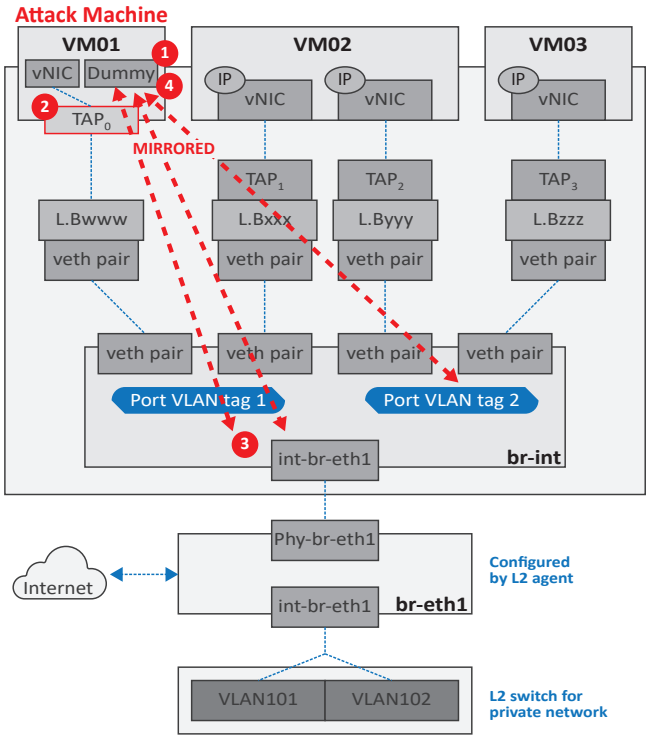


Fig. 6. Attack scenario in OpenStack

and private IP for external network access and internal machine communication, respectively. Target VMs were configured to send between 0 - 15 Kbps to each other. All described experiments were performed 20 times each.

Network setup: VLAN Manager was configured to ensure VM isolation between co-resident VMs by assigning IP addresses and VLAN tag within different ranges known to produce physical resource separation.

VLAN Manager setup: When VLAN mode is enabled, each VM has its own VLAN and network assigned to it. Any physical switches placed inbetween must support 802.1q VLAN tagging for this to function. For correct functioning of the VLAN, the following configuration should be specified in */etc/nova/nova.conf*:

```
network_manager=nova.network.manager.VlanManager
vlan_start=100
dhcpbridge_flagfile=/etc/nova/nova.conf
dhcpbridge=/usr/bin/nova-dhcpbridge
```

Association of public IPs to VMs: A private IP address is automatically assigned when an VM is instantiated. This range of IPs are only accessible within our local environment's network. Public IP addresses are required by the VM to be accessible to the external network. Manual attachment of a public address consists of (1) assigning an address from the available IP range, and (2) linking the address with a VM. Our experiment setup must have a valid range of floating IPs assigned to it for allocation. We do so using nova client:

```
nova floating-ip-create
```

and associating this address to a VM (such as 172.10.1.1).

```
nova add-floating-ip <VM-id> 172.10.1.1
```

This allows for communication with VMs that contain public IP address.

Security configuration: Networking uses iptables to achieve security group functionality, enabling the ipset option to improve security group performance by denoting a hash table. When a new port is created, an additional ipset option is added to the iptables chain. If the security group that the port belongs to contains rules shared by other groups, the group member is added to the ipset chain. If a group member is changed by using ipset, iptables rules are updated instead of being reloaded. Therefore, we initiate a new VM security group by first individually checking for new security group names.

Managing Security Groups: Security groups are configured on the nova-compute host responsible for VM execution, enabling safeguarding of the host machine by limiting access and preventing intrusion of other VMs running on the same host. We launched a security group on port 22. The design of a security group requires two phases, (1) defining a group by using the command `nova secgroup-create`, and (2) setting rules in the group using `nova secgroup-add-rule`:

- For ingress traffic, only traffic matched with security group rules is permitted. All other traffic is dropped, if there is no rule matched.
- For egress traffic, only traffic matched with security group rules is permitted. All egress traffic is dropped, if there is no rule defined.
- When a new security group is created, rules are automatically added to allow/disallow all ingress/egress traffic.

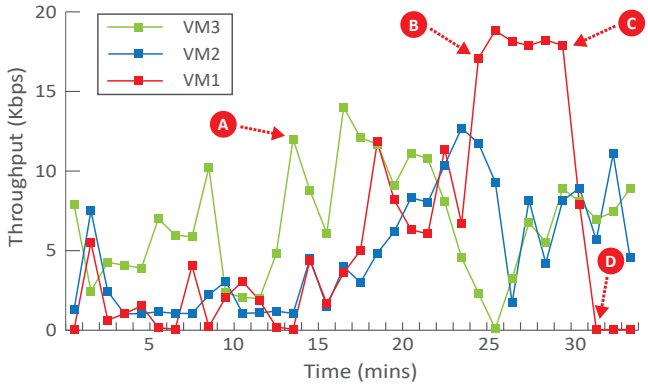
V. Evaluation

A. Network Traffic

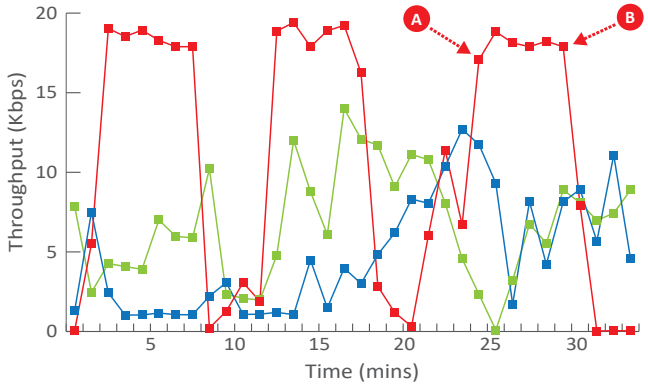
As shown in Figure 7, the attacker VM1 is able to successfully observe traffic between target VMs communicating via the ping command. The attack is effective when it is possible to determine sender and receiver communication between target VMs (e.g., packet header source IP address). If VMs communicate at the same time, the attacker is unable to distinguish the origin of VM traffic.

We studied the subsequent network traffic generated within each VM when an attack is performed. Figure 8(a) shows VM network traffic prior, during, and post attack. Experiment time between 0 - 25 minutes depicts that the attacking VM1 exhibits random network traffic not dissimilar to that of VM2 and VM3 (point A). The attack commences at 25 minutes (point B), where it is observable that the attacking VM1 consumes substantial network traffic compared to target VMs, and continues to do so for 6 minutes until attack completion (point C). The reason for this sudden increase is due to all target VM network traffic being redirected through the attacking VM. When observing network traffic of VM1 in comparison to other VMs during the entire experiment, it is possible that the cloud administrator could detect this as anomalous behavior due to the sudden

Fig. 7. Traffic capturing at attacking VM



(a) Normal co-residing VM network traffic



(b) Cyclical attack pattern network traffic

Fig. 8. Comparison of network traffic flows

spike in network usage. This may lead to further investigation, or deploying a simple countermeasure to restrict VM network traffic that reaches a defined threshold.

However counter measure would be challenging to detect in cloud computing.First - in many public cloud settings, VM resource usage are seen as black boxes by the provider. As long as resource demands do not violate resource capacity requested by a customer, this is seen as typical behavior. Second, even if the system is attempting to monitor atypical resource patterns using bandwidth monitoring tools such as prtg [26], countermeasures will likely include a time delay for determining irregular resource patterns. Therefore, even a few minutes of a VM being compromised may be sufficient for an attacker to achieve their objectives. For example, in 2008 the defense solution of a system operated by the Georgia Government during an HTTP attack [27] activated 5 minutes after the attack had been launched. Finally, if the attacking VM is creating cyclical network patterns prior to an attack, as shown in Figure 8(b), it becomes incredibly difficult to detect.

Within this experiment, attacker VM1 network traffic periodically oscillates between high and low bandwidth, and in the third peak executes an attack following a similar resource pattern seen previously (point A to B).

B. Limitations

The success of our attack is dependent on bridging the TAP interface in the virtual network. Our case study in our experiment describes the current OpenStack network configuration that applies the VLAN and ML2 plug-in for OVS. As a result, this attack only functions on cloud systems using the *neutron* network. The attack does not work on legacy networks such as *nova-network* due to a lack of VLAN and the ML2 plug-in, as well as the network not supporting OVS.

Nova-networking was the only network solution prior to the implementation of Neutron in OpenStack and only supports FLAT network and DHCP services. Flat network or DHCP services followed the same model. The main concept is that all the VMs are attached to the bridge *i.e.* Linux bridge. The bridge is attached to the physical host where there is a support of a physical NIC *i.e.* eth0. Multiple VMs are connected to this bridge. Such limitations are the main cause for cloud providers to adopt advance network module *i.e.* neutron. Table II depicts the different cloud providers vulnerable to this attack.

TABLE II
AN OVERVIEW OF THE VULNERABILITY OF DIFFERENT CLOUD SYSTEMS TO THE PROPOSED APPROACH

Cloud Provider	*Vulnerable to attack*	*Reason*
OpenStack	Yes	Allow bridging of a TAP interface that does not have a private Ethernet interface at backend
Ravello Systems	Yes	Allow bridging of a TAP interface that does not have a private Ethernet interfaces at backend.
Microsoft Azure	No	Prohibited to connect a TAP interface with bridge having no Ethernet at backend.
Google CE	No	Prohibited to connect a TAP interface with bridge having no Ethernet at backend.

VI. Inhibiting the Network-Channel Attack

A. Potential Attack Mitigation Strategies

There exist several relevant works that could be used to mitigate cross-VM attacks. Work from Zhang et al. [2] proposed

to utilize side-channels as a detector to identify illegal co-residency based on the timing channel (accessing L2 cache response time) while Afoulk et al. [28] attempt to avoid conflict of interest among VMs via priority based scheduling.

Another approach would be to directly modify the open source code of OpenStack to limit the granularity of network-based side-channels and penetration of any external device into the current running system. Multiple VM interfaces connect to the OVS internal network bridge, *i.e.* `br-int` that further connects with the physical device and external network. As described above in Section III, the attacker places themselves in the internal network through impersonating the TAP interface which does not have a private Ethernet interface.

As OpenStack code is open source, we were able to identify that the security weakness resides within the networking (neutron) component. As a result, we amended the implementation of neutron in `/opt/stack/neutron/agent/impl_vsctl.py` that includes a method which executes virtual switch operations `def run_vsctl(self,args)` and add a new function `get_all_bridges(self,args)` that collects information of connected interfaces at the bridge. The purpose of this function is to determine whether the connected interface can communicate only using the *TAP* interface, and if capable, block direct connection of all *TAP* interfaces with an OVS bridge accessing the Internet with the same Ethernet. Analysis of the interface reveals that each valid interface has three attributes: - *tag, interface and type*. *Tag* describes that the VLAN is enabled to ensure VM isolation, *interface* its association with back end private Ethernet and *type* interface behaviour. The security check needs to ensure all these attributes of the *TAP* before connecting with the bridge.

VII. Conclusion and Future Work

This research demonstrates a successful cross-VM network attack within OpenStack by a combination of impersonating a TAP interface and constructing a network mirror in the bridge interface. This allows for attackers to successfully redirect and monitor target VM network traffic within the same physical machine unbeknownst to customers. We highlight the challenge for cloud providers to observe and detect such an attack due to an attacking VM not violating requested VM resource capacity, as well as creating cyclical network patterns prior to the attack. Future work will focus on improving the current heuristics that prevent penetrating the external devices into the network. Furthermore, we will investigate how to overcome the challenges of distinguishing between normal resource patterns and target attacks from VMs.

Acknowledgements

This work is supported by the EPSRC (EP/P031617/1).

References

[1] A. Seshadri *et al.*, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 335–350.

[2] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 313–328.

[3] The MITRE Corporation, "CVE-2008-0923," [Accesesd: Jan-18]. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923

[4] T. Ristenpart *et al.*, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *16th ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 199–212.

[5] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX Security Symposium*, 2012, pp. 159–173.

[6] S. T. King *et al.*, *SubVirt: Implementing malware with virtual machines*. IEEE, 2006, vol. 2006, pp. 314–327.

[7] S. R. Kumari and V. Kathiresan, "Virtual environment security-considerations & practices," *Networking and Communication Engineering*, vol. 3, no. 2, pp. 87–92, 2011.

[8] WireShark, [Accessed: Jan-18]. [Online]. Available: https://www.wireshark.org/

[9] OpenStack, "OpenStack Networking ("Neutron")," [Accessed: Jan-18]. [Online]. Available: https://wiki.openstack.org/wiki/Neutron

[10] Aircrack-NG, "Aircrack-NG," [Accessed: Jan-18]. [Online]. Available: https://www.aircrack-ng.org/

[11] H. Wu, Y. Ding, C. Winer, and L. Yao, "Network security for virtual machine in cloud computing," in *International Conference on Computer Sciences and Convergence Information Technology*, Nov 2010, pp. 18–21.

[12] A. Bates *et al.*, "On detecting co-resident cloud instances using network flow watermarking techniques," *International Journal of Information Security*, vol. 13, no. 2, pp. 171–189, 2014.

[13] Y. Zhang *et al.*, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 305–316.

[14] L. H. C. *et al.*, "Analysis on cloud-based security vulnerability assessment," in *IEEE 7th International Conference on E-Business Engineering*, Nov 2010, pp. 490–494.

[15] J. Rutkowska, "Subverting VistaTM kernel for fun and profit," *Black Hat Briefings*, 2006.

[16] S. Murdoch and S. Lewis, "Embedding covert channels into TCP/IP," in *Information Hiding*, vol. 3727. Springer, 2005, pp. 247–261.

[17] P. Ranjith, C. Priya, and K. Shalini, "On covert channels between virtual machines," *Journal in Computer Virology*, vol. 8, no. 3, pp. 85–97, 2012.

[18] S. Hashemi and M. M. Ardakani, "Taxonomy of the security aspects of cloud computing systems-a survey," *networks*, vol. 2, p. 1, 2012.

[19] V. Nirmala, R. K. Sivanandhan, and R. S. Lakshmi, "Data confidentiality and integrity verification using user authenticator scheme in cloud," in *2013 International Conference on Green High Performance Computing (ICGHPC)*, March 2013, pp. 1–5.

[20] S. A. Hussain *et al.*, "Multilevel classification of security concerns in cloud computing," *Applied Computing and Informatics*, vol. 13, no. 1, pp. 57–65, 2017.

[21] OpenStack, [Accessed: Jan-18]. [Online]. Available: http://www.openstack.org

[22] P. England and J. Manferdelli, "Virtual machines for enterprise desktop security," *Information Security Technical Report*, vol. 11, no. 4, pp. 193 – 202, 2006.

[23] M. Piotrowski and A. D. Joseph, "Virtics: A system for privilege separation of legacy desktop applications," UC Berkeley, Tech. Rep. UCB/EECS-2010-70, 2010.

[24] QubesOS, "Qubesos," [Accessed: Jan-18]. [Online]. Available: http://qubes-os.org

[25] A. Marshall *et al.*, "Security best practices for developing windows azure applications," Microsoft Corp., Tech. Rep., 2010.

[26] PRTG, "Bandwidth Monitoring Tool." [Online]. Available: https://https://www.paessler.com/bandwidth_monitoring/

[27] A. Kozlowski, "Comparative analysis of cyberattacks on estonia, georgia and kyrgyzstan," *European Scientific Journal, ESJ*, vol. 10, no. 7, 2014.

[28] Z. Afoulki, A. Bousquet, and J. Rouzaud-Cornabas, "A security-aware scheduler for virtual machines on IaaS clouds," University of Orléans, Tech. Rep. RR-2011-08, 2011.